НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В.Коваль
(підпис) (ініціали, прізвище)
“___”_____2019 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки
6.050103 “Програмна інженерія’’

на тему: Розробка розподіленої віртуальної машини

Виконав: студент 4 курсу, групи ТІ-51

Лучін Андрій Андрійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник старший викладач Ляшенко М. В. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки 6.050103 “Програмна інженерія’’

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2018 р.

ЗАВДАННЯ
на дипломну роботу студенту
Лучіну Андрію Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Розробка розподіленої віртуальної машини”

керівник роботи старший викладач Ляшенко Максим Володимирович

(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”___”_____ 201__ р. № _____

2. Строк подання студентом роботи ___ _____ 201__ р.
3. Вихідні дані до роботи персональний комп’ютер під керуванням операційної системи macOS, мова програмування Go.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Проаналізувати основні проблеми віртуалізації середовиз виконання коду та розподілених систем.
5. Перелік ілюстраційного матеріалу
«Діаграма прецедентів розподіленої віртуальної машини», «Взаємодія процесора, пам’яті, вхідних та вихідних пристроїв», «Операція додавання у регістровій віртуальній машині», «Операція додавання у стековій віртуальній машині», «Інфраструктура розподіленої віртуальної машини», «Діаграма компонентів».

Дата видачі завдання ”___”_____ 201__ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі		
2.	Розробка архітектури та загальної структури системи		
3.	Розробка структур окремих підсистем		
4.	Підготовка матеріалів		
5.	Програмна реалізація системи		
6.	Захист програмного продукту		
7.	Оформлення пояснювальної записки		
8.	Передзахист		
9.	Захист		

Студент _____ Лучін А.А
(підпис) (прізвище та ініціали)

Керівник роботи _____ Кублій Л. І.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Мета роботи — розробка програмного застосунку, що здатен виконувати спеціально розроблений байт-код розподіллено між довільною кількістю вузлів. Для створення програмного забезпечення використано мову програмування Go. Розподілена система побудована на основі протоколу HTTP, з використання стекової віртуальної машини. Для створення тестового середовища з віртуальної мережі використано технологію по управлінню контейнерами Docker та Docker-Compose.

Записка містить 73 сторінки, 11 рисунків, 3 таблиці, 3 додатки і 18 посилань.

Ключові слова: РОЗПОДІЛЕНІ СИСТЕМИ, ВІРТУАЛЬНА МАШИНА, GO, DOCKER, HTTP, JSON, СТЕКОВА ВІРТУАЛЬНА МАШИНА, GRPC, TCP, PROTOBUF, RECURSICE LENGTH PREFIX.

ABSTRACT

The purpose of this work is to develop an application, which is able to execute custom-made byte code in a distributed between arbitrary amount of nodes manner. Distributed system is built upon HTTP, using stack-based virtual machines. For the creation of the test environment Docker and Docker-Compose technologies are used to manage virtual containers.

The note contains 73 pages, 11 figures, 3 tables, 3 attachments and 18 references.

Keywords: DISTRIBUTED SYSTEMS, VIRTUAL MACHINE, GO, DOCKER, HTTP, JSON, STACK-BASED VIRTUAM MACHINE, GRPC, TCP, PROTOBUF, RECURSIVE LENGTH PREFIX.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ЕОМ — Електронна Обчислювальна Машина.

RLP — Recursive Length Prefix.

JSON – JavaScript Object Notation.

PROTOBUF — Protocol Buffers.

TCP — Transmission Control Protocol.

HTTP — HyperText Transfer Protocol.

RPC — Remote Procedural Call.

IDE — Integrated Development Environment

API — Application Programming Interface

GCC — GNU C Compiler

LLVM — Low Level Virtual Machine

JVM — Java Virtual Machine

EVM — Ethereum Virtual Machine

ВСТУП

Значна кількість мов програмування що класифікують як компільовані утилізують алгоритм трансляції вхідного тексту програми у машинний код, що є єдиним інтерфейсом сучасних процесорів для виконання будь-якої логіки. Наприклад:

- C/C++ з інструментарієм GCC, MVSC або ISC
- Rust з інструментарієм Rustc
- Go з інструментарієм gc або gccgo
- Dlang
- Lisp
- Pascal та похідні від нього мови

Основною проблемою підходу компіляції у машинний код є унеможливлення виконання однієї програми у різних середовищах виконання – операційних систем, процесорів, що спричинено відмінністю інтерфейсів операційних систем та наборів машинного коду процесорів. На практиці така особливість трансльованих у машинний код мов значно підвищує кількість ресурсів необхідних для розробки програмного забезпечення, що поширюється на ЕОМ з різним середовищем. Окрім необхідності в повторній компіляції застосунку на кожен тип оточення кінцевого користувача, деякі частини програми, зокрема ті, що використовують специфічні до конкретної операційної системи виклики та переривання, або ж інструкції притаманні лише наборам однієї лінійки процесорів. Зважаючи на життєвий цикл програмного забезпечення, підвищення ресурсної вартості розробки застосунку має силу протягом всього існування продукту. Таким чином, елімінація вищезгаданих проблем може стати дуже ефективною мірою збереження ресурсів.

Одним із перших комерційно успішних проектів, що використовує віртуалізацію середовища виконання програм Java Virtual Machine, розроблена

компанією Sun Microsystems у 1994 році, практично довів, що технологія VM є ефективним рішенням проблеми компільованих мов програмування.

Однак, у зв'язку із розповсюдженням на ринку програмного забезпечення розподілених систем, зокрема Microservices, Embedded Systems та Massively Parallel Systems, виникає необхідність у створенні застосунків, що здатні виконуватися на кількох ЕОМ в межах однієї системи. Існуючі комерційні пропозиції такі, як Oracle Java Virtual Machine або Erlang Virtual Machine не мають такого функціоналу, що б задовольнив попит вищезгаданих розподілених середовищ.

У даній роботі, було досліджено та доведено практичну цінність концепції розподілених віртуальних машин. На відміну від традиційних VM, було реалізовано віртуалізацію повної системи, що складається з кількох ЕОМ замість однієї окремо взятої. Кінцевий продукт дозволяє користувачу розробляти програмне забезпечення, що буде виконуватися розподіллено між машинами-вузлам та впроваджувати його.

Для розробки кінцевого застосунку, було вибрано технології стекових віртуальних машин, що значно спрощує проектування, реалізацію та підтримку програмного забезпечення. Мова програмування Go гарантує ефективність виконання коду, можливість паралелізації та портативність застосунку, тому є очевидним вибором розробки розподіленої віртуальної машини.

Можливі галузі використання: cloud computing, embedded systems, distributed systems, decentralized systems, blockchain and trustless protocols.

1 ЗАДАЧА ВІРТУАЛІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ

Мета розподіленої віртуальної машини – надати користувачу інструментарій, необхідний для виконання програмного коду у віртуальному середовищі розподіллено між декількома ЕОМ під’єднаних до однієї мережі. При цьому, результат виконаних обчислень повинен співпадати на будь-яких платформах, а процес розповсюдження стану віртуальної машини – автоматичним.

Задачу віртуалізації розподілених систем зручно розглядати як ту, що має в собі 2 складові: задача віртуалізації, розподілені системи. Для кожної з них у комерційній та науковій практиці існують установлені методи вирішення. Однією із задач даної роботи – вибрати найбільш оптимальні для кожної складової та продемонструвати їх інтеграцію.

1.1 Віртуалізація

У галузі обчислюваної техніки, віртуалізацією називають створення віртуального, замість актуального, версію чогось, включаючи віртуальне апаратне забезпечення, пристрої для збереження інформації, комп’ютерних мереж [1]. У даній роботі наголос робиться саме на віртуалізацію апаратного забезпечення, зокрема процесора. Це необхідно для гарантування однакової поведінки програмного забезпечення, яке створюється для віртуальної машини, незалежно від операційної системи та процесора ЕОМ на якому виконуються обчислення.

1.2 Розподілені системи

Розподіленими системами називають такі, що складаються с двох або більше компонентів, які мають різні набори функціоналу та обов'язків, розташовані на різних комп'ютерах під'єднаних до мережі, спілкуються між собою за установленим протоколом, та у сукупності досягають спільної мети [2].

Випливаючи із визначення наведеного вище, можна виділити основні структурні елементи розподіленої системи:

- протокол – набір правил, за якими відбувається спілкування по мережі у системі
- компонент – учасник, що приймає вхідні користувацькі дані в систему, повертає вихідні дані користувачу або спілкується с іншими за протоколом

1.3 Вимоги до розподіленої віртуальної машини

Вимоги до задачі даної роботи сформульовано згідно з існуючими проблемами у предметній області, які вона вирішує. Отримані специфікації є основою для проектування кінцевого програмного забезпечення.

По-перше, розподілена віртуальна машина повинна надавати користувачу можливість виконувати написаний ним код у віртуальному середовищі. Результати обчислень повинні повністю співпадати незалежно від платформи, де використано код.

По-друге, користувач повинен мати змогу запускати програмний код розподіллено між вузлами, що розташовані в одній мережі. Необхідно надати механізм для передачі контролю від одного учасника системи до іншого випадково

вибраного, або за переданим ідентифікатором. Важливо взяти до уваги функціональну різність ЕОМ, що беруть участь у протоколі.

По-третє, користувач-розробник повинен мати зручні інструменти для створення, тестування та підтримки програмного забезпечення, що виконується у розподіленій віртуальній машині. Оскільки передбачено велику кількість вузлів, застосунок повинен надавати користувачу можливість працювати з віртуальний середовищем. Інструменти для розробки повинні надавати результати аналогічні до реальної мережі.

На рисунку 1.1 зображено діаграму прецедентів розподіленої віртуальної машини з одним актором – розробник.

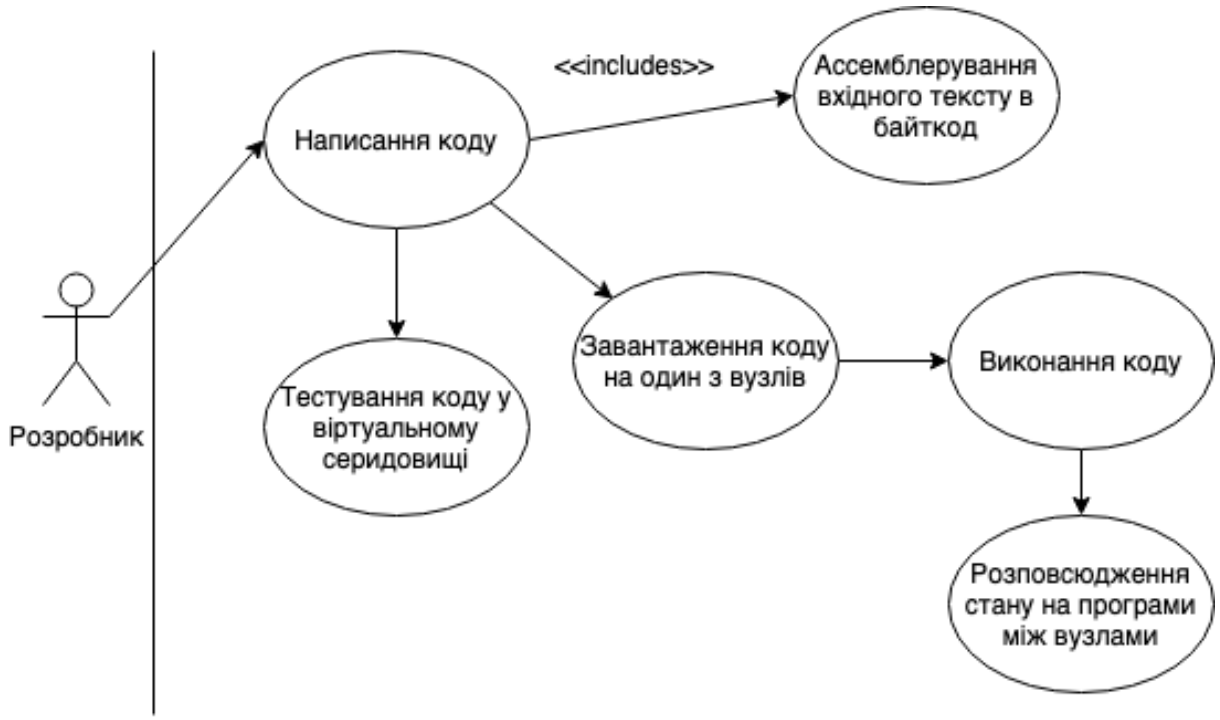

Рисунок 1.1 – Діаграма прецедентів розподіленої віртуальної машини

Інтерфейс для завантаження коду у розподілену віртуальну машину має бути максимально зручним та повним. Ассемблерування вхідного тексту програми повинно відбуватися у фоновому режимі. Розповсюдження стану між вузлами необхідно виконувати автоматично, без участі користувача, але ставивши його до відома.

Однією за найважливіших вимог до розподіленої віртуальної машини є ефективність та швидкість виконання. Оскільки кінцевою платформою для

виконання коду можуть бути вбудовані пристрої побудовані на мікроконтролерах, застосунок повинен бути максимально портативним та працювати в умовах обмежених ресурсів процесора та пам’яті.

Висновки до розділу 1

У даному розділі було сформульовано задачу віртуалізації розподілених систем, здійснено огляд існуючих рішень проблем, що виникають у даній задачі. Було розглянуто концепції віртуалізації та розподілених систем.

2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ВІРТУАЛІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ

На етапі проектування програмного забезпечення дуже важливо прийняти правильні, з огляду на вимоги. У цьому розділі буде порівняно технології важливі для вирішення поставленої задачі - віртуалізація розподілених систем – за рядом спеціально вибраних критеріїв.

2.1 Аналіз алгоритмів віртуальних машин

Для вибору найбільш оптимального для поставленої задачі виконано аналіз. Розглянемо концепцію процесора та пам’яті із перспективи вхідних та вихідних параметрів на рисунку 2.1.

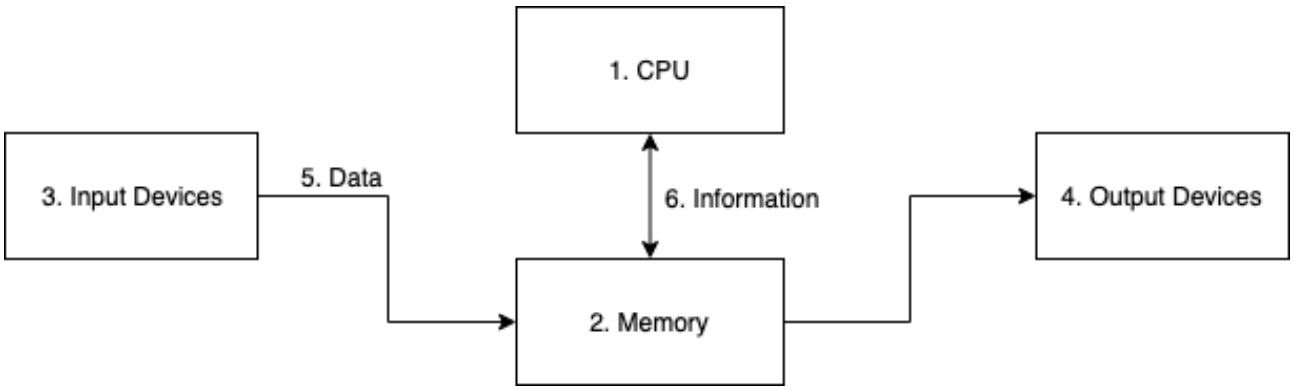

Рисунок 2.1 – Взаємодія процесора, пам’яті, вхідних та вихідних пристроїв

1. Центральний процесор – елемент, що виконує машинні інструкції
2. Пам'ять – елемент, що дозволяє зберігати, записувати та зчитувати інформацію
3. Вхідні пристрої – елементи, що перетворюють користувацькі дії та перетворювати їх в дані. Наприклад: клавіатура, мишка

4. Вихідні пристрої – елементи, що перетворюють дані у деякі події. Наприклад: монітор, динамік, принтер
5. Дані – інформація, що може інтерпретуватися процесором. Наприклад: машинний код, значення змінних
6. Інформація – сигнали та дані, що обмінюються між процесором та пам’яттю під час виконання машинного коду

Для віртуалізації системи, що складається з процесора, пам’яті, вхідних та вихідних пристроїв описаної вище необхідно створити віртуальні аналоги кожного її компонента. Отриманий комплекс називають віртуальною машиною. Існує два основних типи віртуальних машин:

- стекові – ті, що використовують структуру даних стек для зберігання стану
- регістрові – ті, що використовують атомарні клітини пам’яті - регістри

Більшість існуючих комерційних реалізацій віртуальних машин використовують саме стекових підхід [3]. Наприклад: Java Virtual Machine, Erlang Virtual Machine, Python Runtime.

2.1.1 Регістрові віртуальні машини

У реалізаціях, що базуються на алгоритмі регістрових віртуальних машин, структури даних, де зберігаються операнди, базуються на регістрах реальних центральних процесорів. Для запису значень у стан використовують інструкцію MOV, що приймає два аргументи: джерело та призначення. Усі наступні функціональні та арифметичні директиви приймають за аргументи адреси регістрів, що містять необхідні дані. На рисунку 2.2 зображено операцію додавання та значення регістрів до і після виконання.

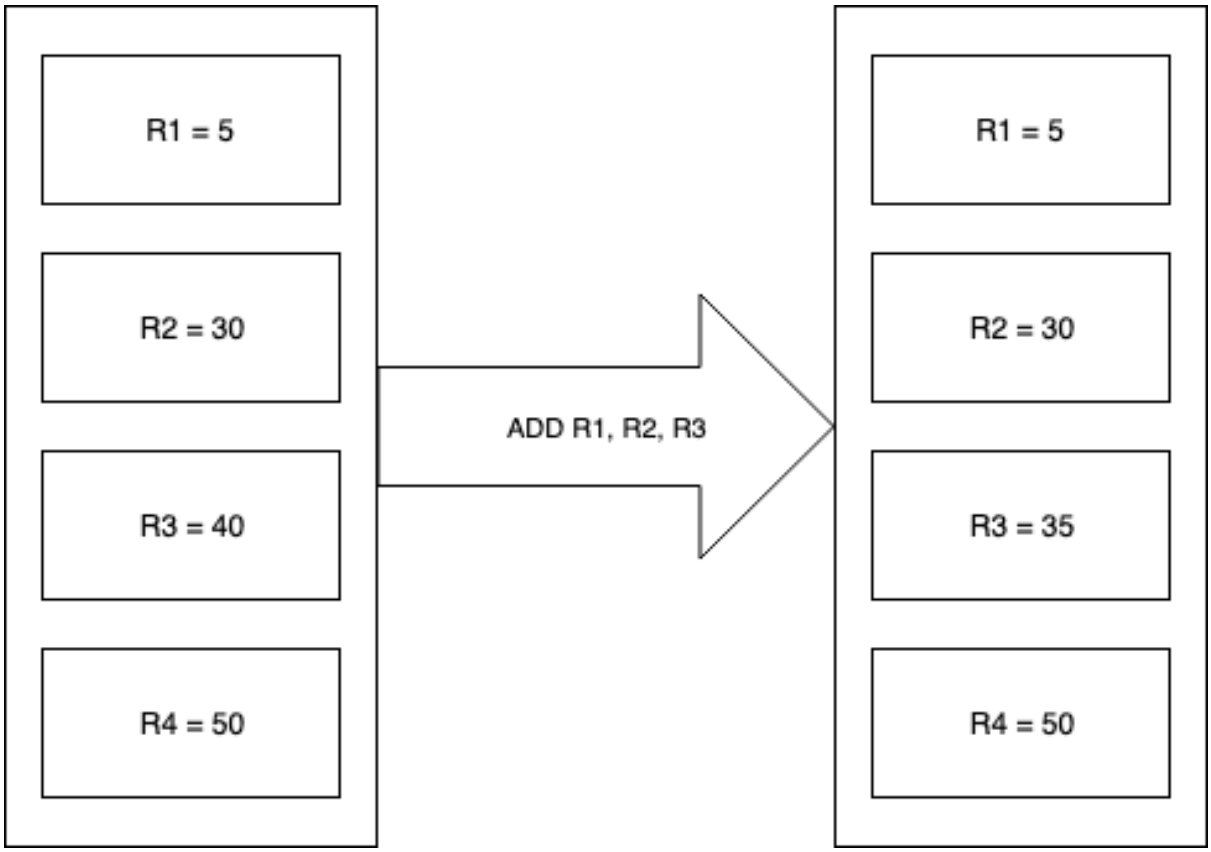

Рисунок 2.2 – Операція додавання у регістровій віртуальній машині

Регістри R1, R2, R3, R4 містять в собі початкові значення. Інструкція ADD R1, R2, R3 слід тлумачити, як «додати значення регістру R1 зі значення R2 та записати результат у R3. На рисунку 2.1 можна побачити як саме змінюється значення стану віртуальної машини.

Однією перевагою регістрової віртуальної машини над стековою – відсутність необхідності витрачати обчислювані ресурси на маніпуляцію стеком. Друга перевага заключається у тому, що регістровий підхід дає більше можливостей для оптимізації вихідного коду. Наприклад: повторні вирази у вхідному коді можна порахувати один раз і зберегти у регістрі, замінюючи наступні виклики дуплікованої ділянки коду на операцію MOV, джерелом-операндом якого буде значення виразу порахованого раніше.

Недоліком регістрової віртуальної машини є те, що її інструкції значно великі, оскільки необхідно передавати значення адрес регістрів. Гарним прикладом є досить розповсюджена директива додавання ADD, що буде приймати щонайменше 2 або 3 аргументи.

2.1.2 Стекові віртуальні машини

Стекові віртуальні машини релізують всі вимоги віртуалізації, наведені у розділі 1.1. Структурую даних для передавання аргументів відповідним інструкціям є стек. Стек — різновид лінійного списку, структура даних, яка працює за принципом «останнім прийшов — першим пішов» [4]. Всі операції в стеку можна проводити тільки з одним елементом, який знаходиться на верхівці стека та був введений в стек останнім. Для зчитування та запису у стан віртуальної машини використовують директиви PUSH та POP. На рисунку 2.3 зображено значення стеку віртуальної машини до та після здійснення операції додавання.

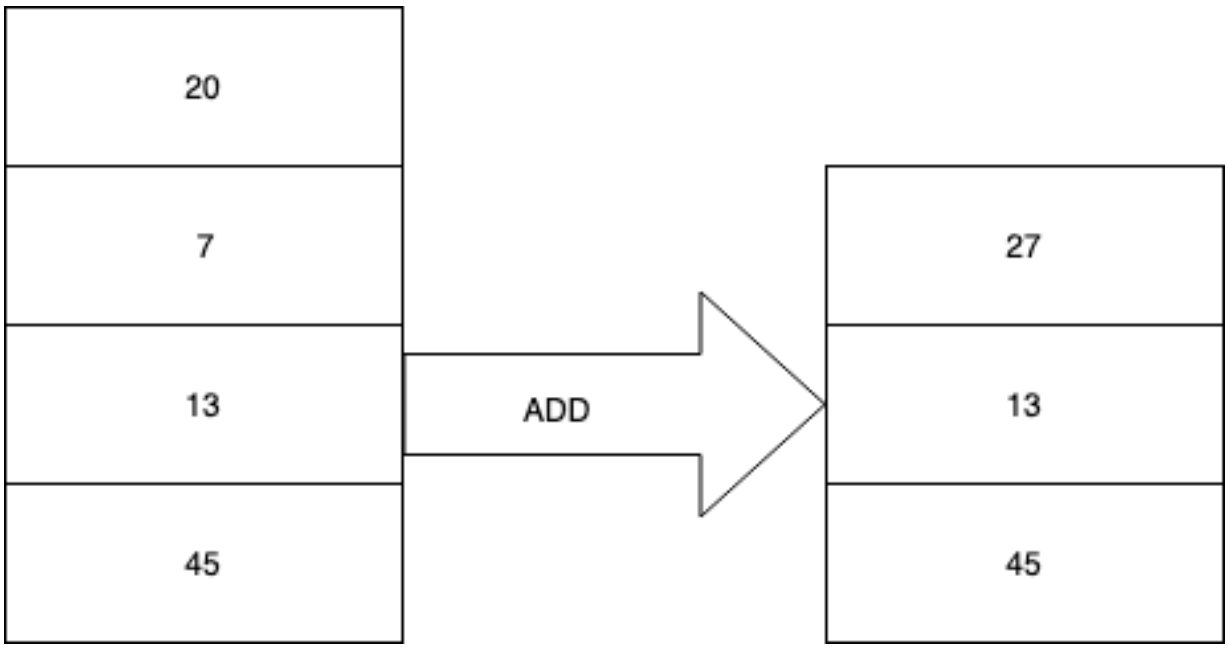

Рисунок 2.3 – Операція додавання у стековій віртуальній машині

До мутації стану, у вершину стеку кладуться значення, які необхідно додати, за допомогою інструкції PUSH. Після виклику директиви ADD два верхні значення стеку видаляються та їхня сума кладеться до вершини стеку. За таким же принципом передаються аргументи до інших інструкцій віртуальної машини.

Перевагою стекового підходу є можливість оминути значення адреси операндів, так як вони за визначення знаходяться на вершині структурі даних [5]. Це дозволяє значно зменшити розмір інструкцій. Більш того, стекові віртуальні машини потребують значно менших ресурсів для імплементації ніж регістрові, що дозволяє надавати програмному забезпеченню більш швидку підтримку у разі виникнення помилок, якісніше його документувати та розширювати.

2.2 Аналіз протоколів спілкування розподілених систем

Від вибору способу комунікації між вузлами розподіленої віртуальної машини буде залежить ефективність програмного забезпечення, швидкість виконання, змога розширювати програмне забезпечення додатковим функціоналом та розмір даних, які можна буде пересилати по мережі [6]. Розглянемо деякі існуючі у науковій та комерційній практиці протоколи:

1. HTTP – скорочено з HyperText Transfer Protocol, використовується у всесвітньому павутинні як основний протокол спілкування між браузером та сервером.
2. TCP – скорочено з Transmission Control Protocol - низькорівневий протокол, що є основою для більшості інших. Створений для гарантування збереження цілісності даних при передачі ненадійним транспортом
3. gRPC – скорочено з Google Remote Procedural Call – реалізація Remote Procedural Call створена компанією Google
4. Websocket – протокол для двосторонньої тривалої комунікаціями між двома вузлами

Як правило, протоколи ділять за можливістю використання у кінцевих програмних застосунків на низькорівневі та високорівневі. Однак, для кожної

системи слід використовувати такий, що найбільше відповідає її вимогам, зокрема швидкість виконання, обмеження з розміру переданих даних чи портативність.

2.2.1 Протокол HTTP

Протокол передачі гіпертексту, або HTTP – це високорівневий протокол для розподілених колаборативних інформаційних систем, що використовуються для передачі даних за «клієнт-серверною» моделлю. Існує два типи повідомлень у протоколі HTTP: запит та відповідь, що робиться клієнтом та сервером відповідно. Учасників мережі знаходять та ідентифікують за допомогою URL – Universal Resource Locator, більш відомі як посилання.

Кожен запит/відповідь складається з трьох частин:

1. стартовий рядок;
2. заголовки;
3. тіло повідомлення, що містить дані запиту, запитаний ресурс або опис проблеми, якщо запит не виконано.

Обов'язковим мінімумом запиту є стартовий рядок. Починаючи з HTTP/1.1 обов'язковим став заголовок Host: (щоб розрізнити кілька доменів, які мають одну й ту ж IP-адресу). Протокол HTTP дозволяє клієнту встановлювати наступні типи запиту для додавання контексту к переданим даним:

- OPTIONS - повертає методи HTTP, які підтримуються сервером. Цей метод може служити для визначення можливостей веб-сервера
- GET - запитує вміст вказаного ресурсу. Запитаний ресурс може приймати параметри. Вони передаються в рядку URI. Згідно зі стандартом HTTP, запити типу GET вважаються ідемпотентними — багатократне повторення одного і того ж запиту GET повинне приводити до однакових результатів

- HEAD - аналогічний методу GET, за винятком того, що у відповіді сервера відсутнє тіло.
- POST - передає призначені для користувача дані заданому ресурсу. При цьому передані включаються в тіло запиту. На відміну від методу GET, метод POST не вважається ідемпотентним
- PUT - завантажує вказаний ресурс на сервер
- PATCH - завантажує певну частину ресурсу на сервер
- DELETE - видаляє вказаний ресурс
- TRACE - повертає отриманий запит так, що клієнт може побачити, що проміжні сервери додають або змінюють в запиті
- CONNECT - для використання разом з проксі-серверами, які можуть динамічно перемикатися в тунельний режим SSL

Відповідь сервера складається зі статусу, що містить код та опис результату виконання клієнтського запиту, та тіла, що містить певні дані які повертаються клієнту [7]. Код статусу може свідчити про успішне виконання або помилку. Наприклад:

- 1xx — інформаційний: запит прийнятий, продовжуй процес.
- 2xx — успіх: дія була успішно передана, зрозуміла, та прийнята.
- 3xx — перенаправлення: наступні дії мають бути успішно виконані для реалізації запиту.
- 4xx — помилка клієнта: запит містить синтаксичні помилки або не може бути виконаний.
- 5xx — помилка сервера: сервер не зміг виконати правильно сформований запит.

Таким чином, за допомогою клієнт-серверної моделі спілкування, типів запитів та статусами відповідей, HTTP дозволяє створювати комплексні системи застосунків, що знаходяться в мережі та стандартизує їх взаємодію. Така абстракція дозволяє витрачати менше ресурсів на розробку форми спілкування та сконцентрувати їх на

створення бізнес-логіки, що значно підвищує якість програмного забезпечення за рахунок пришвидшення підтримки та розширення існуючого функціоналу [8]. У зв’язку з тим, що HTTP широко використовується у всесвітньому павутинні та інших комерційних мережах, можна використовувати оптимізовані та якісно протестовані імплементації у вигляді бібліотек, що розповсюджуються за відкритими ліцензіями.

2.2.2 Проткол TCP

Протокол контролю трансмісії, або TCP, - є низькорівневим, стрижневим протоколом Інтернету [15]. Протокол призначений для управління передачею даних у комп'ютерних мережах, працює на транспортному рівні моделі OSI. На відміну від іншого поширеного протоколу транспортного рівня UDP, TCP забезпечує надійне доправляння даних від хоста-відправника до хоста-отримувача, для цього встановлюється логічний зв'язок між хостами. Таким чином TCP належить до класу протоколів зі встановленим з'єднанням.

Незважаючи на відсутність функціоналу притаманного високорівневим протоколам, які можна використовуваті для побудови кінцевих програмних застоснуків, TCP широко використовуються у комерційних розподілених системах для побудови поверх нього нових протоколів [9]. Таких підхід необхідний у системах, вимоги яких не може задовільнити ні один з існуючих на ринку рішень, або через їх недоступність на кінцевій платформі програмного забезпечення. Оскільки TCP є стрижневим протоколом мережі Інтернет, його імплементації вбудовані у всі сучасні операційні системи та платформи, що мають можливість спілкуватися мережею.

Значним недоліком TCP є його обмежений функціонал, що робить необхідним розширювати його власними ресурсами розробника програмного забезпечення [16]. Також, із-за алгоритма, що називають congestion, TCP дуже неефективно працює в умовах ненадійної або повільної мережі.

2.2.3 Протокол gRPC

Протоколи віддаленого процедурного виклику, або RPC, використовують звичну програмним інженерам абстракцію функцій та процедур для опису взаємодії учасників мережі. Схема RPC заохочує імплементації притримуватися моделі клієнт-сервер, проте не обмежується нею [10]. Суть взаємодії закладається у пересиланні на севрер запиту, що містить назву методу що необхідно виконати та параметри які йому передаються. У відповідь клієнт отримує результат виконання запитаної процедури.

Одна з імплементацій RPC – gRPC, розроблена компанією Google, поширюється за відкритою ліцензією, що дає можливість використовувати її за межами компанії-виробника у комерційних цілях. Взаємодія між клієнтом і сервером у даній реалізації називається сервісом та описується мовою Protobuf, де вказується сигнатура методів, що клієнт може викликати у сервера. Протокол заохочує користувачів використовувати інструменти для автоматичної генерації коду на основі опису сервісу, що дозволяє підтримувати цілісність реалізації в учасниках системи.

Перевагою протоколу gRPC є його висока ефективність, швидкість та експресивність. У зв’язку з цим, його широко використовують у системах, де швидкість реагування має дуже важливе значення.

Одним з найголовнішим недоліком gRPC є його слабка портативність. Реалізація протоколу можлива лиш на обмеженій кількості платформ із-за особливостей своєї побудови, що базується на HTTP/2 та Protobuf. Також, слід пам’ятати, що gRPC дуже тісно зв’язаний з вбудованим форматом серіалізації, що робить неможливим його заміну у разі необхідності.

2.3 Порівняння проколів спілкування по мережі

Наведені вище мережеві протоколи порівняно за наступними критеріями:

- наявність високорівневого функціоналу: типізованих повідомлень, помилок тощо
- наявність якісних реалізацій з відкритою ліцензією на кілька мов програмування
- тип комунікації: атомарний - одне повідомлення за раз - чи продовжене – дані передаються протягом існування підключення
- зв’язаність з форматом кодування даних
- гарантія цілісності переданих даних

Таблиця 2.1 показує порівняння протоколів HTTP, TCP та gRPC за вказаними вище характеристиками.

Таблиця 2.1 – Порівняння протоколів HTTP, TCP та gRPC

Критерій	HTTP	TCP	gRCP
Високорівневий функціонал	Присутній	Відсутній	Присутній
Реалізації з відкритою ліцензією	Присутні	Присутні	Присутні
Тип комунікації	Атомарний	Продовжений	Атомарний
Зв’язаність з форматом кодування даних	Не існує	Не існує	Існує
Цілісність переданих даних	Гарантується	Гарантується	Гарантується

Усі порівняні протоколи гарантують цілісність передачі даних та мають реалізації з відкритими ліцензіями, проте значення інших критеріїв не збігаються.

2.4 Формати серіалізації

Дані які передають мережею використовуючи протоколи кодують у спеціальний бінарний формат, який вибирається з огляду на вимоги програмного забезпечення. У деяких випадках формат серіалізації прив’язаний до протоколу, як це можна спостерігати з gRPC та виглядом бінарних даних Protobuf. Розглянемо деякі існуючі у науковій та комерційній практиці алгоритми серіалізації:

- JSON – JavaScript Object Notation, історично використовувся для кодування об’єктів у мові програмування JavaScript, проте еволюціонував у один з найбільш популярних форматів спільного призначення
- Protobuf – алгоритм розроблений компанією Google, тісно зв’язаний з gRPC. Виділяється компактністю закодованих даних та вбудованою мовою для визначення типів даних
- RLP – Recursive-Length-Prefix – широко використовується у децентралізованих системах, зокрема в системі побудованій на технологію Blockchain Ethereum. Відомий компактністю даних та простотою реалізації. Має всього 2 типи: цілі числа та списки
- Bencode – формат серіалізації, що використовуються протокол BitTorrent для кодування torrent файлів. Має всього 4 типи даних: строки, цілі числа, списки та словники

Від вибору алгоритму кодування залежить розмір даних, які будуть пересилатися по мережі, здібність розширювати на підтримувати програмне забезпечення та швидкість роботи системи [11]. Варто пам’ятати тісний зв'язок формату серіалізації та протоколу та вибирати перший з огляду на інший.

2.4.1 Формат JSON

Нотація об’єктів JavaScript, або JSON, - текстовий формат даних, створений для кодування значень об’єктів у мові програмування JavaScript, проте не залежить від неї. На таблиці 2.2 перераховано типи даних, що підтримуються JSON.

Таблиця 2.2 – Типи даних формату JSON

Тип даних	Опис
Number	Застосовується для кодування чисельних даних. Підтримує цілі та дійсні числа. Розмір закодованого значення необмежений
String	Застосовується для кодування текстових даних. Обов’язковий алгоритм кодування – UTF-8, що дозволяє використовувати літери будь-якого алфавіту
Boolean	Застосовується для зберігання булевих значень – true або false
Array	Масив застосовується для кодування декількох атомарних значень. Формат JSON не потребує однакового типу для всіх елементів масиву
Object	Обєкт являє собою словник, та використвоується для збергіання даних виду ключ-значення, де ключем є строка, а значення будь-який з типів даних JSON

Формат даних JSON поєднує в собі зручність для використання людиною із-за свого текстового вигляду та простоту реалізації і експресивністю, що робить його зручним для використання комп'ютерними програмами. Дані закодовані цим форматом можна архівувати утилітою gzip, що значно зменшує розмір кінцевого кодування. Реалізації серіалізації JSON вбудовані в стандартні бібліотеки більшість популярних мов програмування, включаючи Java, C++, Go, Python, PHP, Kotlin, Swift, JavaScript.

2.4.2 Формат Protobuf

Формат кодування Protobuf розроблений компанією Google та розповсюджується за відкритою для комерційного користування ліцензією [12]. Даний алгоритм архітектурно зв'язаний з протоколом віддалених процедурних викликів gRPC, проте його можливо використовувати окремо.

Структури кодованих даних в Protobuf описуються спеціальною мовою. Це дозволяє з мінімальною затратою ресурсів дуплікувати структури з мови програмування, що використовується для створення програмного забезпечення.

Алгоритм кодування Protobuf підртимує велику кількість скалярних типів, включаючи цілі та дійсні числа різної довжини, строки та булеві значення. Для організації складених типів дозволяється використовувати списки та словники.

2.4.3 Формат RLP

Основна мета RLP – кодування довільно вкладених масивів бінарних даних. Функціонал серіалізації більш високорівневих структур даних таких, як строки, словники, списки залишається застосункам, що використовують RLP [13]. Даний

формат не є людино-читаємим, проте із-за своєї тривіальною структури та компактністю закодованих даних є оптимальним вибором серед форматів серіалізації, що використовуються виключно програмним забезпеченням.

Формат RLP має всього два типи даних:

1) Число
2) Масив

Розмір масиву кодується в якості префіксу до даних. Якщо розмір префіксу перевищує допустимий, рекурсивно додається ще один.

Висновки до розділу 2

У даному розділі було проаналізовано два алгоритми віртуальних машин, три протоколи спілкування розподілених систем та три формати кодування даних. Кожен має свої переваги та недоліки, які необхідно брати до уваги при виборі для застосування у програмному забезпеченні.

3 ОПИС ПОБУДОВАННОЇ МОДЕЛІ РОЗПОДІЛЕНОЇ ВІРТУАЛЬНОЇ МАШИНИ

У даному розділі описано модель побудованого застосунку для віртуалізації розподілених систем ти методи розв’язання поставленої задачі. Програмний застосунок побудовано на основі вимог сформульованих у першому розділі даної роботи.

3.1 Огляд вибраних методів

У другому розділі роботи було проаналізовано методи розв’язання поставлених задач, а саме: алгоритми віртуальної машини, протоколи мережевого спілкування та формати кодування даних. На основі отриманої інформації вибрано найбільш оптимальні для розподіленої віртуальної машини.

Віртуальну машину реалізовано за стековим підходом. Оскільки фундаментальною особливість проекту є передача стану між вузлами, необхідно мінімізувати розмір даних, що поширюються мережею. Завдяки цьому, отриманий застосунок буде працювати швидше, адже мережі у реальному світі ненадійні та передача даних по них дуже повільна, порівняно з виконанням обчислень центральним процесором. У стековій віртуальній машині розмір інструкцій значно менший ніж у регістрових, а її стан є монолітним та більш компактним. Відповідно даний підхід являється найбільш оптимальним.

Протоколом розподіленої системи вибрано HTTP. Високорівневий функціонал, що є частиною проколу дозволяє створювати кінцеві застосунки з мінімальною затратою ресурсів, при цьому є душе швидким та надійним у порівнянні з gRPC та TCP. Атомарний характер даних повідомлень співпадає з форматом повідомлень в розподіленій віртуальній машині. Типи запитів на статус-коди дозволяють семантично доповнити передані дані метаінформацією.

Формат JSON вибраний як основний алгоритм серіалізації стану віртуальної машини, що передається по мережі між вузлами за протоколом HTTP [14]. Завдяки великій кількості підтримуваних типів формат задовольняє вимоги з кодування програмних об’єктів. Наявність реалізацій з відкритою ліцензією дозволяє впровадити у розподілену віртуальну машину оптимізований та протестований код. Окрім того, текстовий вигляд формату JSON інтегрується з тілом HTTP запитів.

3.2 Інфраструктура розподіленої системи

Вибраний тип інфраструктури основується на концепції рівноправних вузлів. Програмний застосунок, який називається агентом, встановлюється на всі ЕОМ, що є учасниками системи. Подальший функціонал роботи виконується агентом, включаючи ассемлерування вхідного тексту програми, виконання отриманого байт-коду та розповсюдження стану віртуальної машини іншим вузлам. На рисунку 3.1 зображено отриману інфраструктуру.

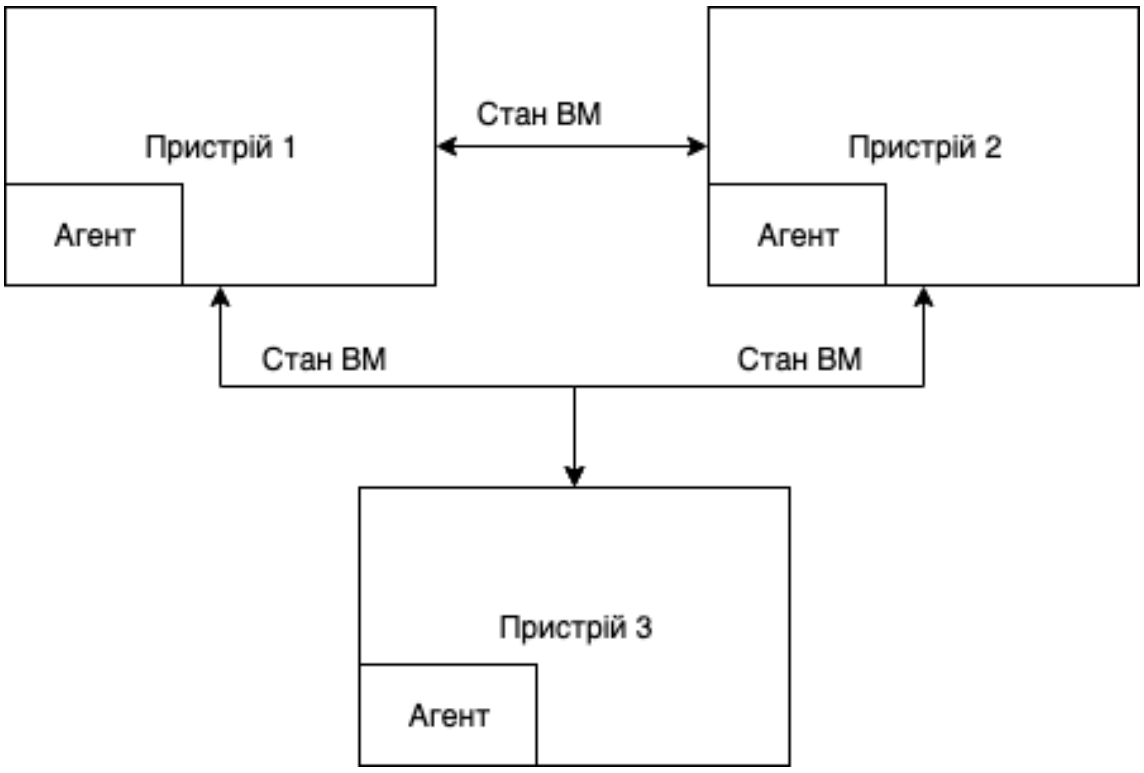

Рисунок 3.1 – Інфраструктура розподіленої віртуальної машини

Отримана схема взаємодії дозволяє легко розширювати систему додатковими вузлами. Для цього необхідно тільки встановити агент на нову ЕОМ та змінити конфігурацію системи.

3.3 Компоненти програмного застосунку

Застосунок розподіленої віртуальної машини складається з трьох основних компонентів:

1) Віртуальна машина
2) Протокол спілкування між вузлами
3) Ассемблер

На рисунку 3.2 зображено UML діаграму компонентів.

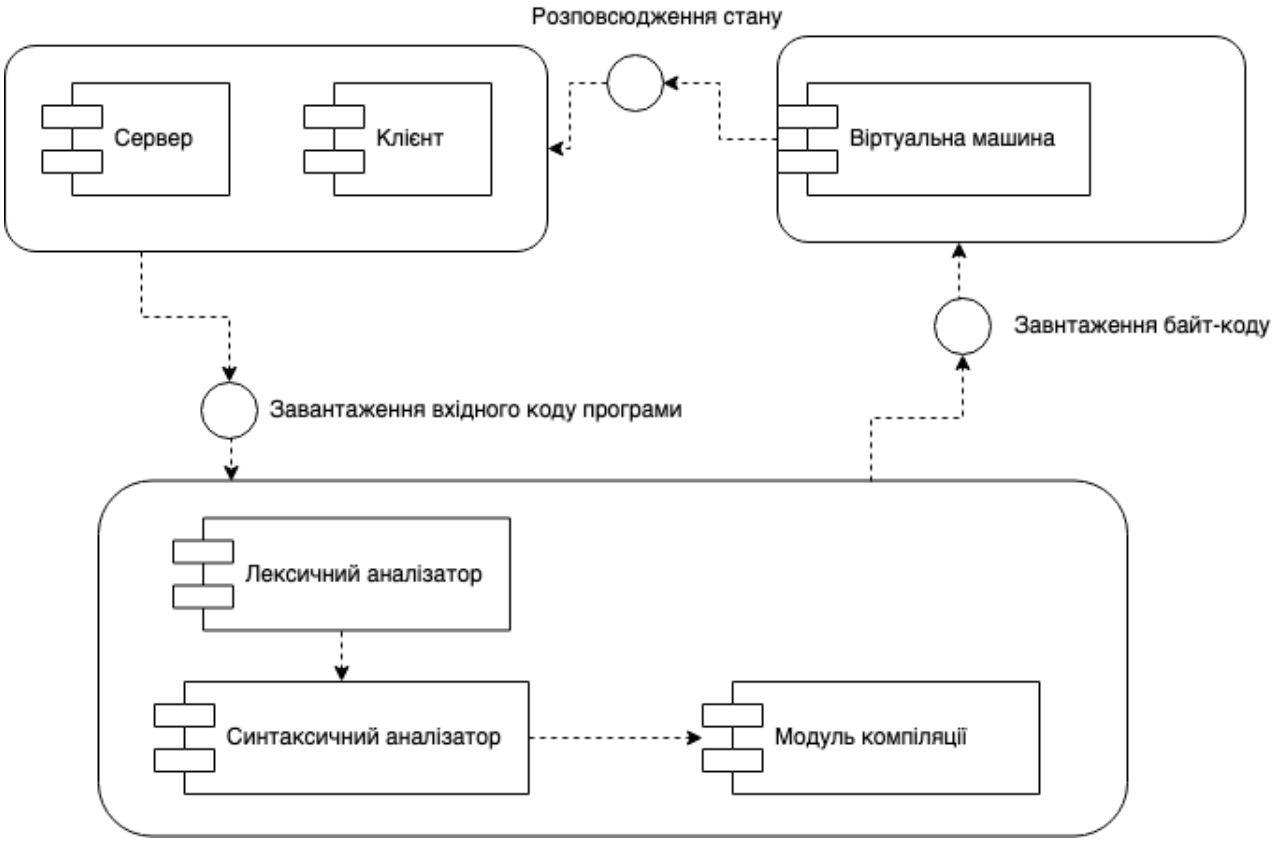

Рисунок 3.2 – Діаграма компонентів

Під час проектування компонентів використано один із шаблонів проектування GRASP – Loose Coupling, що передбачає мінімальну взаємодію між компонентів [15]. Кожна частина програми отримує дані лише від однієї іншої частини, та відправляє результат наступній.

Компонент віртуальної машини включає в себе реалізацію інструкцій, інтерпретатор байт-коду та серіалізацію стану віртуальної машини. Вхідні дані до цього компоненти – список інструкцій, вихідні – результат виконаної програми.

3.4 Тестове середовище

Для тестування програм, що створюються для віртуальної машини, дана робота пропонує зручний спосіб для розгортання віртуальної інфраструктури на одній ЕОМ. Технологія Docker використовується для створення малих за розміром контейнерів на основі обраної користувачем архітектури процесора та операційної системи. Інструмент Docker-compose дозволяє створити довільну кількість віртуальних ЕОМ зі встановленим заздалегідь агентом розподіленої системи та автоматично під’єднує їх до спільної мережі. За допомогою одної консольної команди створюється тестове середовище у якому виконується програма.

Висновки до розділу 3

У даному розділі описано модель програмного застосунку розподіленої віртуальної машини та технології, що використовуються – стекові віртуальні машини, HTTP, JSON. Наведено інфраструктуру отриманої розподіленої системи та компоненти програмного продукту. Описано метод вирішення проблеми необхідності тестового середовища.

4 ЗАСОБИ РОЗРОБКИ

Для реалізації розподіленої віртуальної машини використано мову програмування Go. В якості інтегрованого середовища розробки обрано застосунок GoLand розроблений компанією JetBrains.

4.1 Середовище GoLand

GoLand є кодовою назвою нового комерційного інтегрованного середовища розробки компанії JetBrains, спрямованого на створення ергономічної середовища для розвитку мови програмування Go. Нова IDE розширює платформу IntelliJ за допомогою інтеграції інструментів, специфічних для мови Go.

При встановленні GoLand, потрібно ознайомитись та взяти до уваги вимоги до апаратного забезпечення, а саме:

- Мінімальна оперативна пам'ять 2 ГБ, рекомендована оперативна пам'ять 4 Гб
- Об'єм жорсткого диска 1,5 Гб, а також принаймні 1 ГБ для кеш-пам'яті
- Мінімальна роздільна здатність екрана 1024x768

Щодо вимог до програмного забезпечення, вони є наступними:

- JRE 1.8 поєднується з дистрибутивом GoLand.

Goland розповсюджується як TAR GZ архів на системах GNU Linux, як MSI інсталятор Microsoft Windows 10 та APP пакет на Apple MacOS. Інструкції інсталяції залежать від платформи.

Інтегроване середовище розробки компанії JetBrains, GoLand, підтримується на операційних системах вказаних на таблиці 4.1.

Таблиця 4.1 – Рекомендовані вимоги та підтримувані операційні системи

Windows	macOS	Linux
32-розрядні або 64-розрядні версії Microsoft Windows 10, 8, 7 (SP1) або Vista (SP2)	macOS 10.8.3 або пізнішої версії (підтримуються лише 64-розрядні системи)	• ОС Linux (зверніть увагу, що 32-розрядний JDK не постачається в комплекті, тому рекомендується 64-бітна система) • Рекомендовано робоче середовище KDE, Gnome або Unity

При першому запуску GoLand, після завершення початкової конфігурації GoLand з'являється вікно вітання зображене на рисунку 4.1.

Рис. 4.1. Привітальне вікно GoLand

При налаштуванні робочої області, перш ніж почати створення чи імпортування проекту GoLand, потрібно переконатися, що зроблено наступне:

1) Створено робочий простір, де зберігаються проекти.

2) Завантажено розповсюдження Go.

3) Визначено змінну середовища GOPATH, яка вказує на розташування робочої області. Ця змінна буде використовуватися командами Go та IDE для вирішення операторів імпорту, встановлення, створення та оновлення пакетів тощо.

Створюючи новий проект, потрібно:

4) Натиснути "Створити новий проект" на екрані привітання.

5) Якщо робоче середовище налаштовано, GoLand запропонує створити проект у srcsubdirectory. Вказати назву вашого проекту в полі Location, Go SDK (GOROOT) і натиснути Create. GoLand створює проект у робочій області та робить відповідні налаштування Go.

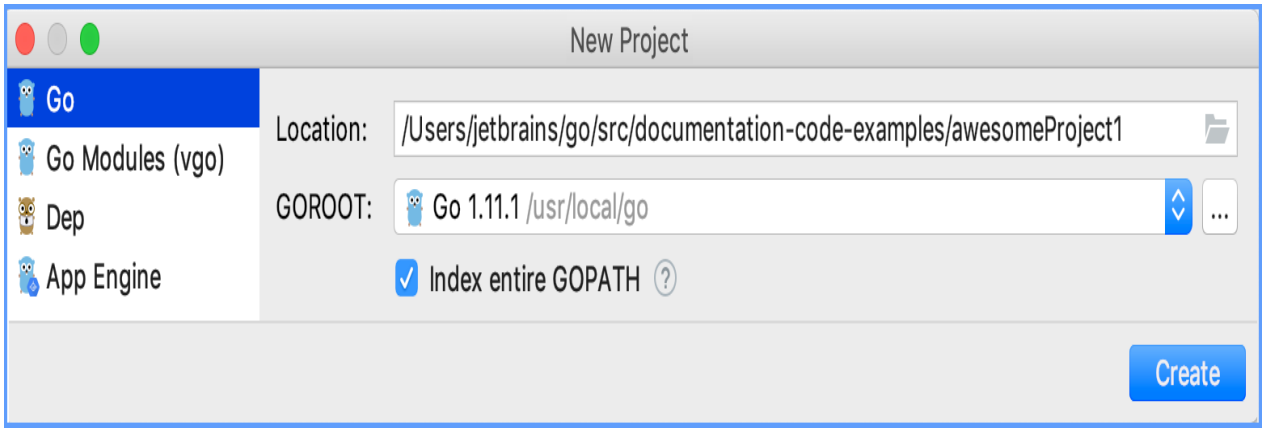

Рис 4.2. Створення нового проекту в інтегрованому середовищі GoLand

Для поточного запуску програми, потрібно зробити наступне:

- У вікні інструмента «Project» натиснути правою кнопкою миші проект.
- У контекстному меню виберіть New, Go File.
- У діалоговому вікні, що відкриється, вкажіть назву вашої програми та виберіть Simple application.

Щоб запустити код Go, потрібно налаштувати конфігурацію запуску або створити нову. Ви можете керувати вашими конфігураціями запуску у діалоговому вікні Run/Debug.

1) У головному меню виберіть Run .
2) У діалоговому вікні Run/Debug конфігурації виберіть відповідну конфігурацію.
3) Вкажіть параметри конфігурації виконання. Можна також вказати параметр Before Launch, наприклад, можна додати команду, яка буде виконатись, перед виконанням створеної конфігурації. Натисніть кнопку OK.
4) У головному меню виберіть Run або Run <configuration_name> або натисніть Ctrl + Shift + F10

Однією з особливостей інтерфейсу користувача GoLand є редактор, оскільки він дозволяє викликати практично будь-яку функцію IDE, не виходячи з нього, що допомагає організувати макет, у якому є більше простору екрана, оскільки допоміжні елементи керування, такі як панелі інструментів і вікна, приховані.

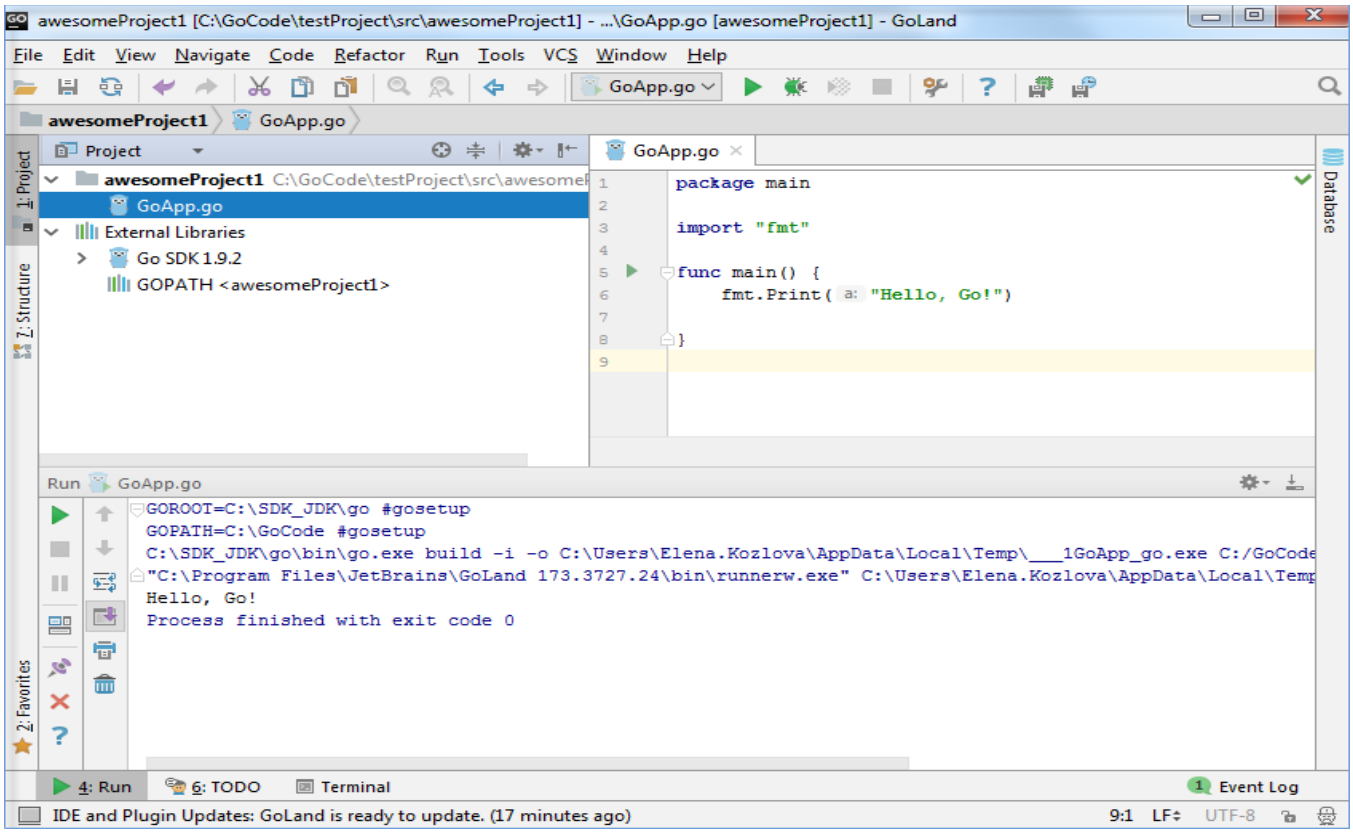

Рис. 4.3. Редактор GoLand

Розумне завершення, яке допомогає написати новий код за допомогою автоматичного заповнення операторів. Ctrl + Shift + Space надає вам список найбільш відповідних символів, які застосовуються в поточному контексті. Коли ви вибираєте пропозицію.

Також, корисною функцією є перевірка коду та швидкі виправлення. Інтегроване середовище розробки забезпечує вбудовані перевірки, які перевіряють код, коли ви його вводите. Як тільки вони знаходять проблематичний код, вони надають швидкі виправлення, які можна застосувати, просто натиснувши Alt + Enter.

Доступний рефакторинг включає перейменування і вилучення, що дозволяє швидко і безпечно змінювати код.

Достатньо лише один клік для переходу до супер-методу, реалізації, використання, декларації, до будь-якого класу, файлу або символу, будь-якої дії або вікна інструменту.

Семантичне виділення розширює стандартне підсвічування синтаксису унікальними кольорами для кожного параметра та локальної змінної.

Іншою корисною функцією є підказки параметрів. Редактор показує підказки параметрів для літералів та нульових значень, які використовуються як аргументи методу. Ці підказки роблять код набагато більш читабельним.

GoLand включа в себе повнофункціональний відладчик, який підтримує його загальні функції: годинник, оцінювати вираз, показувати вбудовані значення та інше. Відладчик працює як для додатків, так і для тестів.

Також, GoLand забезпечує спеціальний інтерфейс для запуску і налагодження тестів та перевірок.

Щодо фронт-енд і бек-енд розробки, інтегроване середовище розробки успадковує від WebStorm свою першокласну підтримку для фронтальних мов і фреймворків. Воно пропонує першокласну допомогу для кодування JavaScript,

TypeScript, Dart, React та багатьох інших мов. Підтримка Angular та Node.js доступна за допомогою плагінів.

Також, GoLand підтримує інтелектуальне кодування під час редагування SQL; підключення до живих баз даних; виконання запитів; переглядати та експортувати дані; і навіть керувати своїми схемами у візуальному інтерфейсі - прямо з інтегрованого середовища розробки.

Нарешті, більше 10 років розробки платформи IntelliJ надають більше 50+ плагінів GoLand різної природи, включаючи підтримку додаткових VCS, інтеграції з різними інструментами та фреймворками, а також удосконалення редактора, такі як емуляція Vim [16].

Отже, проаналізувавши всі аспекти інтегрованного середовища розробки GoLand, можна виділити наступні переваги:

- Golang має потужний вбудований двигун автозаповнення
- має вбудований відладчик. Точки перерви допомагають вам зрозуміти і використовувати паралелізм
- працює на Windows, Mac і GNU / Linux
- fmt файл/проект, файл Goimports, файл Gogenerate, каталог Gotype, файл Vet Go
- має вбудовані клієнти контролю версій як для git, так і для mercurial
- рефакторінг
- вбудований командний рядок
- безкоштовно для студентів

Так само, інтегроване середовище Goland маж наступні недоліки:

- закрита ліцензія, ціна

- використання пам'яті. Goland 1.0 Preview використовує 650 Мб оперативної пам'яті на Windows 10
- Час запуску може зайняти деякий час

Таким чином, програмний застосунок Goland є оптимальним вибором для інтегрованого середовища у проекті розподіленої віртуальної машини. Недоліки можна вважати незначними у порівнянні з широким списком переваг.

4.2 Мова програмування Go

Для реалізації розподіленої віртуальної машини використано мову програмування Go. Go розроблений компанією Google та розповсюджується за відкритою ліцензією MIT, що передбачає вільне та безкоштовне використання у будь-яких сферах діяльності, включаючи комерційну та академічну.

Мова програмування була створено з наголосом на швидкість компіляції вхідного коду, виконання отриманого програмного забезпечення та простоту створення застосунків [17]. Основною рисою Go є наявність у мові інструментів для написання паралельного коду та синхронізації – Goroutines та Channels. Завдяки цьому, можна створювати дуже ефективні паралельні алгоритми, що обмінюються інформацією каналами.

Програмний код написаний мовою Go ділиться на пакети – packages. Надалі їх можна використовувати повторно у інших застосунках. Імпорт бібліотек виконується методом вказання гіперсилки на Git репозиторій необхідного пакету. Пошук необхідних бібліотек можна здійснювати на централізованих регістрах, наприклад Awesome Go.

Мова програмування Go має велику та оптимізовану стандарту бібліотеку. Вона містить в собі реалізації наступних алгоритмів:

- криптографічні

- кодування даних JSON
- протоколи HTTP та HTTP/2
- математичні функції
- протоколи TCP та UDP
- кодування Unicode UTF-8 та UTF-16
- роботи з часом та датою

Для програмних застосунків, що потребують асинхроне спілкування по мережі, Go пропонує використання Goroutine у парі з реалізованими алгоритмами. Це дозволяє створювати оптимальні додатки, швидкість виконання яких може зрівнятися з низькорівневими інструментами [18].

Окрім мови програмування, існує велика кількість інструментів для форматування коду, налагодження помилок, статичних та динамічних аналізаторів. Наприклад: gofmt, goretruns, godebug.

Висновки до розділу 4

В даному розділі було розглянуто засоби, що використовувалися для розробки програмного застосунку розподіленої віртуальної машини. Створене забезпечення використовує мову програмування Go та інтегровану систему розробки GoLand.

5 ПРОГРАМНА РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ ВІРТУАЛЬНОЇ МАШИНИ

У даному розділі роботи продемонстровано використання вибраних технологій та алгоритмів для вирішення поставленої задачі, описані деталі реалізації програмних компонентів у кінцевому застосунку.

5.1 Віртуальна машина

Компонент віртуальної машини включає в себе структуру даних, або стан, методи, що оперують над станом, програмні інструкції та кодування стану віртуальної машини. Структура даних складається з:

- стек – для зберігання стану виконання програми
- середовище – для запису даних типу ключ-значення
- значення instruction pointer – вказує на теперішнє положення в програмі
- код програми – список інструкцій, які необхідно виконати

Віртуальна машина має інтерфейс для мутації або зчитування будь-якого елемента структури. Дані методи використовуються інструкціями для емуляції діяльності процесора. У розподіленій віртуальній машині реалізовані наступні інструкції:

- push – для запису значення в стек
- pop – для зчитування зі стеку
- put – для запису в середовище
- get – для зчитування з серидовища
- nop – нульова інструкція
- print – для виведення значення в консоль

- add – для додавання двух чисел

Кожна інструкція має спеціально призначений операційний код, що використовується для кодування в бінарний формат. За необхідності користувач може розширити список інструкцій. Це корисно у ситуаціях коли один з вузлів має унікальний функціонал, наприклад є єдиним ЕОМ під'єднаним до зовнішньої мережі, або є датчиком рівня світла. Код до віртуальної машини подається у вигляді списка інструкцій, який формується іншими компонентами застосунка.

5.2 Ассемблер

Для інтерпретації програми у текстовому вигляді у роботі реалізовано ассемблер. Він приймає вхідний код та формує список інструкцій, який надсилається до віртуальної машини. Алгоритм ассемблерування складається з лексично та синтаиксичного аналізів і процесу компіляції. Граматика мови ассемблера виглядає наступним чином:

<програма> ::= <інструкція> | <програма> NEWLINE <інструкція>

<інструкція> ::= <назва> <аргументи>

<назва> ::= nop | push | pop | get | put | print | yield

<аргументи> ::= ∅ | <аргумент> | <аргументи>, <аргумент>

<аргумент> ::= <ціле> | <ідентифікатор>

<ціле> ::== <цифра> | <ціле> <цифра>

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ідентифікатор> ::= <літера>| < ідентифікатор> <літера>

<літера> ::= a…z | A…Z

5.2.1 Лексичний аналіз

Для лексичного аналізу вибрано алгоритм кінцевого автомату. Аналізатор приймає код програми у текстовому вигляді та формує лекскеми, які у подальшому надсилаються на синтаксичний аналіз. Ассемблер розподіленої віртуальної машини розділяє наступні лексеми:

- ключові слова NOP, PUSH, POP, GET, PUT, PRINT, YIELD
- ідентифікатор IDENT
- цілі значення INTEGER
- розділові знаки COMMA, NEWLINE, EOF

Отримані лексеми кодуються спеціальним двійковим форматом, що складається з коду лексеми та позиції в програмі.

5.2.2. Синтаксичний аналіз

Для синтаксичного аналізу розроблено власний алгоритм, що є більш оптимальним для даної задачі ніж існуючі. Аналіз виконується по одній інструкції за раз. Тип директиви рахується за початковим ключовим словом. Якщо перша лексема рядка не є ключовим словом повертається помилка. В залежності від наявності та кількості зчитуються аргументи для кожної інструкції та записуються у відповідну до типу інструкції структуру даних. Вхідними даними синтаксичного аналізу є список лексем, вихідним – список об’єктів структур даних, що містять тип та значення аргументів.

5.2.3 Компіляція

Вхідні даних до компілятора є список структур згенеровані синтаксичним аналізатором. Вихідними – байт-код, що розуміється віртуальною машиною. Для кодування використовують операційні коди об’явлені у модулі віртуальної машини.

Приклад програми написаній на мові ассемблер:

```
push 12
push 27
add
print
yield
```

Дана програма додає числа 12 та 27, виводить результат в консоль та передає контроль наступному вузлу.

5.3 Протокол спілкування між вузлами

Для спілкування між вузлами використовується HTTP. Кожен вузол має обробник запитів за адресою /yield, що приймає стан віртуальної машини та передає його до віртуальної машини. Окрім цього, віртуальна машина має змогу використовувати клієнт протоколу для передачі контроля над виконанням програми наступному вузлу. Це реалізовано за допомогою HTTP запитів, тілом якого є стан закодований у форматі JSON.

Другим обробником запитів є точка доступу для надсилання ассемблерного коду у текстовому вигляді, що розташована за адресою /assemble. Отримане тіло запиту направляється до компоненту ассемблера.

Кожен вузол має доступ до списку IP адрес кожного іншого вузла. Цей список можна змінювати, додаючи нових учасників системи. Адреси використовуються застосунком автоматично для надсилання повідомлень усім агентам у мережі.

Висновки до розділу 5

У даному розділі роботи продемонстровано алгоритми лексичного та синтаксичного аналізів, компіляції, віртуальної машини та реалізацію протоколу, що використовуються у розподіленій віртуальній машині.

6 МЕТОДИКА ВИКОРИСТАННЯ РОЗРОБЛЕНОЇ ТЕХНОЛОГІЇ

Існує два способи для використання розподіленої віртуальної машини: у реальному та тестовому середовищі. Для розробки та тестування застосунків поверх системи слід використовувати тестове середовище.

6.1 Створення тестового середовища

Для формування тестової мережі із віртуальних ЕОМ використовуються технології Docker та Docker-Compose. На рисунку 6.1 зображено опис контейнера, в який встановлюється агент розподіленої віртуальної машини.

```
# Build-base
FROM golang:1.12-alpine AS build_base

RUN apk add --no-cache git

WORKDIR /dvm

ENV GO111MODULE=on

COPY go.mod .
COPY go.sum .

RUN go mod download

# Build stage
FROM build_base AS builder

RUN apk add --no-cache git gcc musl-dev

WORKDIR /dvm

ADD . .
RUN go build -o dvm

# Run stage
FROM alpine:latest

RUN apk add --no-cache ca-certificates
COPY --from=builder /dvm/dvm /dvm/bin/
ADD ./*.json /dvm/bin/

WORKDIR /dvm/bin

EXPOSE 80
ENTRYPOINT ["./dvm"]
```

Рисунок 6.1 – Опис контейнера вузла

Контейнер складається з шару побудови програмного застосунку для забезпечення кешування артефактів збірки та етапу запуску агенту. Для бази контейнера використовуються операційна система GNU Linux Alpine, проте її можна змінити на будь-яку необхідну користувачу. Виконуваний файл лінкується зі своїми залежностями статично, щоб гарантувати успішний запуск на всіх платформах.

Між створеними контейнерами будується віртуальна мережа. Кожен вузол має доступ до іншого за допомогою спеціального текстового ідентифікатора, що використовується замість IP адреси. При використанні, ідентифікатор за допомогою DNS перетворюється в реальну адресу контейнера. На рисунку 6.2 зображено опис побудованої системи.

```yaml
version: "3"

services:
  alpha:
    build:
      context: .
    ports:
      - "80:80"
    networks:
      main:
        aliases:
          - alpha
    command: alpha.json

  beta:
    build:
      context: .
    networks:
      main:
        aliases:
          - beta
    command: beta.json

  gamma:
    build:
      context: .
    networks:
      main:
        aliases:
          - gamma
    command: gamma.json

networks:
  main:
```

Рисунок 3.2 – Опис мережі контейнерів

Тестова мережа створює три вузли: alpha, beta, gamma з встановленим за запущеним агентом. Після виклику команди docker-compose up розподілена віртуальна машина готова для використання.

6.2 Створення реального середовища

Для створення системи на реальних ЕОМ необхідно спочатку збудувати, налаштувати конфігурацію та запустити агент на кожному вузлі. Це можно зробити за допомогою команди go build -o dvm, що створе виконуваний файл dvm. Далі створити файл-конфігурацію з розширенням json, в якому перераховані IP адреси інших вузлів. Формат файлу конфігурації зображено на рисунку 6.3.

```
{
  "peers": [
    "http://alpha:80",
    "http://gamma:80"
  ]
}
```

Рисунок 6.3 – Формат файлу конфігурації

Опісля, необхідно запустити виконуваний файл, передавши як аргумент шлях до конфігурації. Наприклад ./dvm beta.json.

6.3 Використання розподіленої віртуальної машини

Першим етапом є написання ассемблерного коду, що необхідно виконати на розподіленій віртуальній машині та зберегти його до файлу з розширенням dvm. Після цього, використовуючи будь-який зручний HTTP клієнт надіслати код на один

з вузлів. Слід використовувати обробник /assemble. Далі розподілена віртуальна машина автоматично виконає заданий код, передаючі контроль між вузлами де необхідно. Результат виконаних операцій можна спостерігати на самих вузлах. У разі виникнення синтаксичний чи інших помилок сервер поверне відповідь з кодом 400.

Висновки до розділу 6

У даному розділі описано алгоритм розгортання тестового або реального середовища, та використання його для обчислення програм.

ВИСНОВКИ

Сформульовано задачу віртуалізації розподілених систем, здійснено огляд існуючих рішень проблем, що виникають у даній задачі. Було розглянуто концепції віртуалізації та розподілених систем.

Проаналізовано стекові та регістрові алгоритми віртуальних машин. Продемонстровано алгоритм роботи кожної із них. Виконано порівняльний аналіз за спеціально вибраними критеріями та обґрунтовано вибір найбільш оптимального.

Порівняно протоколи спілкування по мережі HTTP, TCP та gRPC. Наведено список функціоналу, сфери використання та технічні характеристики кожного з них. Проаналізовано формати кодування даних JSON, RLP та Protobuf за рядом критеріїв. Описано взаємодію кожного з алгоритмів серіалізації з протоколами.

Описано проект реалізованого програмного застосунку. Наведено список компонентів, їх відповідальності та взаємодії. Ілюстровано архітектуру інфраструктури розподіленої системи.

Виявлено найбільш оптимальні алгоритми для лесичного, синтаксичного аналізів, компіляції, виконання та розповсюдження програмного коду. Розроблено ассемблерну мову програмування, що використовується віртуальною машиною.

Реалізовано протокол спілкування розподілених систем на основі HTTP та формату JSON. Описано отриманий додаток, обробники запитів та схему взаємодії.

Проаналізована використані технології для розробки кінцевого програмного продукту, наведені переваги та недоліки кожної із них. На основі виконаного аналізу, вибрано найбільш оптимальні, продемонстровано їх роботу.

Продемонстровано алгоритм створення реальної та тестової мережі використовуючи технологію Docker. Наведено приклад програми та метод її впровадження у розподілену віртуальну машину.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гультяев, А.К. Виртуальные машины: несколько компьютеров в одном [Текст]: учебник / Гультяев А.К. - Санкт - Петербург: Издательский дом «Питер», 2006. — С. 22-23.
2. Таренбаум Э. Распределенные системы. Принципы и парадигмы. –СПб: Питер, 2003 – 877с.
3. Гобрик, Д. А. Человеко-машинный интерфейс / Д. А. Гобрик; науч. рук. И. И. Гутич // Материалы 70-й студенческой научно-технической конференции : [тезисы докладов студентов БНТУ] / ред. колл.: Е. Е. Трофименко [и др.]. — Минск: БНТУ, 2015. — С. 45-46.
4. Басс Л., Клементс П., Кацман Р. Архитектура программного обеспечения на практике. 2-е изд. Л. : Питер, 2010.
5. Буч Г. Об’єктно-орієнтоване проектування з прикладами застосування / Г. Буч. — К.: Академія, 2002. — 723 с.
6. Генри С. Уоррен — Алгоритмические трюки для программистов / Генри С. Уоррен. — М.: Вильямс, 2014. — 512 с.
7. Теленик С.Ф., Ролік О.І., Букасов М.М. Моделі управління розподілом обмежених ресурсів в інформаційно-телекомунікаційній мережі // Вісник НТУУ «КПІ». Інформатика, управління та обчислювальна техніка. — К.: Екотех. — 2006. — №44. — С. 243—246.
8. David Marshall, Wade A. Reynolds, Dave McCrory. Advanced Server Virtualization: VMware and Microsoft Platforms in the Virtual Data Center. — Auerbach Publications, 2006. — 760 p.
9. Стенин А.А., Тимошин Ю.А., Шемсединов Т.Г. Метод динамической интерпретации метамоделей в разработке прикладных информационных систем. – Материалы международной научно- практической конференции “Академическая наука – проблемы и достижения”.-М.-2012. – сс. 186–192.

10. Хантер Р. Основные концепции компиляторов. – М.: Вильямс, 2002. – 256 с.
11. JSON [Електронний ресурс]. – Режим доступу: https://www.json.org/
12. Ксензов М.В. Рефакторинг архитектуры программного обеспечения. – М.: ИСП РАН. – Препринт 4, 2004. – 12 с.
13. Богданов В.Л., Гордєєв В.С. Практический опыт написания синтаксического анализатора языка программирования Кобол - http://se.math.spbu.ru/reeng.html.
14. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. – М.: Издательский дом «Вильямс», 2003. – 768 с.
15. Рейуорд-Смит В. Дж. Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988. – 128 с.
16. Ларман К. Применение UML и шаблонов проектирования. – М.: Издательский дом «Вильямс», 2001. – 496 с.
17. Jin C. FAST TCP: Motivation, architecture, algorithms, performance / C. Jin, D.X. Wei, S.H. Low. // In IEEE INFOCOM 2004. – Hong Kong, March 2004.
18. Xu L. Binary increase congestion Control (BIC) for Fast Long-Distance Networks / L. Xu, K. Harfoush, I. Rhee // Proceedings of IEEE INFOCOM 2004. – Hong Kong, March 2004.

ДОДАТОК А

Технологія взаємодії застосунків MATLAB і C# у комплексі моделювання гідроакустичних процесів

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51181_18Б

Аркушів 2

Київ 2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ТІ51181_18Б	Записка.docx	Текстова частина дипломної роботи
Компоненти		
УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ ТІ51181_18Б 12-1	dvm.exe	Основний компонент додатку розподіленої віртуальної машини

ДОДАТОК Б

Розробка розподіленої віртуальної машини

Текст програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51181_18Б

Аркушів 11

Київ 2019

```
package lex

import (
	"fmt"
	"unicode"
)

const eof = -1

const (
	commentChar = ';'
	newLineChar = '\n'
	commaChar   = ','
)

const (
	nopText   = "nop"
	pushText  = "push"
	putText   = "put"
	getText   = "get"
	printText = "print"
	addText   = "add"
	yieldText = "yield"
)

type TokenType int

const (
	Nop TokenType = iota
	Push
	Put
	Get
	Print
	Add
	Yield
	Ident
	Comma
	Integer
	NewLine
	EofToken
)

func Lex(text string) ([]Token, error) {
	lexer := newLexer(text)
	for s := onStart(&lexer); s != nil; s = s(&lexer) {
	}

	if lexer.err != nil {
		return nil, lexer.err
	}
	return lexer.result, nil
}

func (t TokenType) String() string {
	switch t {
	case Nop:
		return nopText
	case Push:
		return pushText
	case Put:
		return putText
	case Get:
		return getText
	case Print:
		return printText
	case Add:
		return addText
	case Yield:
		return yieldText
	case Ident:
		return "ident"
	case Comma:
		return ","
	case Integer:
		return "integer"
	case NewLine:
		return "newLine"
	}
	return ""
```

```
}

type state func(*lexer) state

type Token struct {
	Typ TokenType
	Val string
}

func (t Token) String() string {
	return fmt.Sprintf("%s %s", t.Typ, t.Val)
}

type lexer struct {
	text   []rune
	pos    int
	acc    []rune
	result []Token
	err    error
}

func newLexer(text string) lexer {
	return lexer{
		text: []rune(text),
	}
}

func (l *lexer) next() rune {
	if l.pos >= len(l.text) {
		return eof
	}

	l.pos++
	return l.text[l.pos-1]
}

func (l *lexer) peek() rune {
	if l.pos >= len(l.text) {
		return eof
	}

	return l.text[l.pos]
}

func (l *lexer) skipLine() {
	for ; l.text[l.pos] != newLineChar; l.pos++ {
	}
	// Skip the newline character.
	l.pos++
}

func (l *lexer) accVal() string {
	return string(l.acc)
}

func (l *lexer) pushAcc(c rune) {
	l.acc = append(l.acc, c)
}

func (l *lexer) clearAcc() {
	l.acc = l.acc[:0]
}

func (l *lexer) emit(tok Token) {
	l.result = append(l.result, tok)
}

func (l *lexer) emitErr(err error) {
	l.err = err
}

func onStart(l *lexer) state {
	switch c := l.next(); {
	case c == commentChar:
		l.skipLine()
		return onStart
	case unicode.IsLetter(c):
		l.pushAcc(c)
		return onAtom
```

```
		case unicode.IsDigit(c):
			l.pushAcc(c)
			return onInteger
		case c == newLineChar:
			l.emit(Token{Typ: NewLine})
			return onStart
		case c == commaChar:
			l.emit(Token{Typ: Comma})
			return onStart
		case unicode.IsSpace(c):
			return onStart
		case c == eof:
			l.emit(Token{Typ: EofToken})
			return nil
		default:
			l.emitErr(unexpectedChar(c, "beginning of the token"))
			return nil
		}
}

func onAtom(l *lexer) state {
		switch c := l.peek(); {
		case unicode.IsLetter(c):
			l.pushAcc(c)
			l.next()
			return onAtom
		case unicode.IsDigit(c):
			l.pushAcc(c)
			l.next()
			return onAtom
		case isDelim(c):
			tok := makeOperator(l.accVal())
			if tok == nil {
				l.emit(Token{Typ: Ident, Val: l.accVal()})
				l.clearAcc()
				return onStart
			}

			l.emit(*tok)
			l.clearAcc()
			return onStart
		default:
			l.emitErr(unexpectedChar(c, "atom"))
			return nil
		}
}

func onInteger(l *lexer) state {
		switch c := l.peek(); {
		case unicode.IsDigit(c):
			l.pushAcc(c)
			l.next()
			return onInteger
		case isDelim(c):
			l.emit(Token{Typ: Integer, Val: l.accVal()})
			l.clearAcc()
			return onStart
		default:
			l.emitErr(unexpectedChar(c, "integer"))
			return nil
		}
}

func isDelim(c rune) bool {
		return unicode.IsSpace(c) || c == commaChar || c == commentChar
}

func makeOperator(text string) *Token {
		switch text {
		case nopText:
			return &Token{Typ: Nop}
		case pushText:
			return &Token{Typ: Push}
		case putText:
			return &Token{Typ: Put}
		case getText:
			return &Token{Typ: Get}
		case printText:
			return &Token{Typ: Print}
```

```
		case addText:
			return &Token{Typ: Add}
		case yieldText:
			return &Token{Typ: Yield}
		}

		return nil
}

func unexpectedChar(char rune, where string) error {
		var repr string
		if char == '\n' {
			repr = "new line"
		}
		repr = "character `" + string(char) + "`"

		return fmt.Errorf("unexpected %s when parsing %s", repr, where)
}
package parse

import (
		"fmt"
		"strconv"

		"github.com/pkg/errors"

		"github.com/andreyluchin/dvm/asm/lex"

		"github.com/andreyluchin/dvm/vm"
)

func Parse(tokens []lex.Token) ([]vm.Instr, error) {
		parser := newParser(tokens)
		for s := startParse(&parser); s != nil; s = s(&parser) {
		}

		if parser.err != nil {
			return nil, parser.err
		}

		return parser.result, nil
}

type parser struct {
		tokens []lex.Token
		pos    uint
		acc    []lex.Token
		line   uint
		result []vm.Instr
		err    error
}

type state func(*parser) state

func newParser(tokens []lex.Token) parser {
		return parser{
			tokens: tokens,
			line:   1,
		}
}

func (p *parser) next() lex.Token {
		p.pos++
		return p.tokens[p.pos-1]
}

func (p *parser) peek() lex.Token {
		return p.tokens[p.pos]
}

func (p *parser) pushAcc(tok lex.Token) {
		p.acc = append(p.acc, tok)
}

func (p *parser) clearAcc() {
		p.acc = p.acc[:0]
}

func (p *parser) emit(instr vm.Instr) {
```

```
		p.result = append(p.result, instr)
}

func (p *parser) emitErr(err error) {
		p.err = err
}

func (p *parser) parseNewLine() error {
		tok := p.next()
		if tok.Typ != lex.NewLine {
				return unexpectedToken(tok, p.line)
		}

		p.line++
		return nil
}

func startParse(p *parser) state {
		switch tok := p.peek(); tok.Typ {
		case lex.Nop:
				return parseNop
		case lex.Push:
				return parsePush
		case lex.Put:
				return parsePut
		case lex.Get:
				return parseGet
		case lex.Print:
				return parsePrint
		case lex.Add:
				return parseAdd
		case lex.Yield:
				return parseYield
		case lex.NewLine:
				p.next()
				return startParse
		case lex.EofToken:
				return nil
		default:
				p.emitErr(unexpectedToken(p.peek(), p.line))
				return nil
		}
}

func parseNop(p *parser) state {
		nop := p.next()
		if nop.Typ != lex.Nop {
				p.emitErr(unexpectedToken(nop, p.line))
				return nil
		}
		err := p.parseNewLine()
		if err != nil {
				p.emitErr(err)
				return nil
		}
		p.emit(&vm.Nop{})
		return startParse
}

func parsePush(p *parser) state {
		push := p.next()
		if push.Typ != lex.Push {
				p.emitErr(unexpectedToken(push, p.line))
				return nil
		}

		val := p.next()
		if val.Typ != lex.Integer {
				p.emitErr(unexpectedToken(val, p.line))
				return nil
		}

		err := p.parseNewLine()
		if err != nil {
				p.emitErr(err)
				return nil
		}

		valInt, err := strconv.Atoi(val.Val)
```

```
		if err != nil {
			err := errors.Wrap(err, "error when parsing an integer")
			p.emitErr(err)
			return nil
		}

		p.emit(&vm.Push{
			Val: vm.Val(valInt),
		})

		return startParse
}

func parsePut(p *parser) state {
		put := p.next()
		if put.Typ != lex.Put {
			p.emitErr(unexpectedToken(put, p.line))
			return nil
		}

		key := p.next()
		if key.Typ != lex.Ident {
			p.emitErr(unexpectedToken(key, p.line))
			return nil
		}

		err := p.parseNewLine()
		if err != nil {
			p.emitErr(err)
			return nil
		}

		p.emit(&vm.Put{
			Name: vm.Key(key.Val),
		})

		return startParse
}

func parseGet(p *parser) state {
		get := p.next()
		if get.Typ != lex.Get {
			p.emitErr(unexpectedToken(get, p.line))
			return nil
		}

		key := p.next()
		if key.Typ != lex.Ident {
			p.emitErr(unexpectedToken(key, p.line))
			return nil
		}

		err := p.parseNewLine()
		if err != nil {
			p.emitErr(err)
			return nil
		}

		p.emit(&vm.Get{
			Name: vm.Key(key.Val),
		})

		return startParse
}

func parsePrint(p *parser) state {
		print := p.next()
		if print.Typ != lex.Print {
			p.emitErr(unexpectedToken(print, p.line))
			return nil
		}

		err := p.parseNewLine()
		if err != nil {
			p.emitErr(err)
			return nil
		}

		p.emit(&vm.Print{})
```

```
	return startParse
}

func parseAdd(p *parser) state {
	add := p.next()
	if add.Typ != lex.Add {
		p.emitErr(unexpectedToken(add, p.line))
		return nil
	}

	err := p.parseNewLine()
	if err != nil {
		p.emitErr(err)
		return nil
	}

	p.emit(&vm.Add{})

	return startParse
}

func parseYield(p *parser) state {
	yield := p.next()
	if yield.Typ != lex.Yield {
		p.emitErr(unexpectedToken(yield, p.line))
		return nil
	}

	err := p.parseNewLine()
	if err != nil {
		p.emitErr(err)
		return nil
	}

	p.emit(&vm.Yield{})

	return startParse
}

func unexpectedToken(tok lex.Token, line uint) error {
	return fmt.Errorf("unexpected token `%v` when parsing line #%d", tok, line)
}
```

```
package asm

import (
	"fmt"

	"github.com/andreyluchin/dvm/asm/lex"
	"github.com/andreyluchin/dvm/asm/parse"
	"github.com/andreyluchin/dvm/vm"
	"github.com/pkg/errors"
)

// Assemble turns text into vm instructions.
func Assemble(text string) ([]vm.Instr, error) {
	tokens, err := lex.Lex(text)
	if err != nil {
		return nil, errors.Wrap(err, "error when lexing the program")
	}

	fmt.Printf("Tokens:\n%v\n", tokens)

	res, err := parse.Parse(tokens)
	if err != nil {
		return nil, errors.Wrap(err, "error when parsing the program")
	}

	return res, nil
}
```

```
package protocol

import (
	"bytes"
	"encoding/json"
	"fmt"
	"net/http"
```

```
	"github.com/pkg/errors"

	"github.com/andreyluchin/dvm/vm"

	log "github.com/sirupsen/logrus"
)

const yieldMethod = "yield"
const assemleMethod = "assemble"

type Peer interface {
	fmt.Stringer
	Yield(state *vm.VM) error
}

func NewHttpPeer(url string) Peer {
	return httpPeer{
		url: url,
	}
}

type httpPeer struct {
	url string
}

func (p httpPeer) String() string {
	return p.url
}

func (p httpPeer) Yield(state *vm.VM) error {
	stateBytes, err := json.Marshal(state)
	log.Infof("%s\n", string(stateBytes))
	if err != nil {
		return errors.Wrap(err, "error when encoding state")
	}
	stateReader := bytes.NewReader(stateBytes)

	path := fmt.Sprintf("%s/%s", p.url, yieldMethod)
	resp, err := http.Post(path, "application/json", stateReader)
	if err != nil {
		return errors.Wrap(err, "error when making http request")
	}

	if resp.StatusCode < 200 && resp.StatusCode >= 300 {
		return fmt.Errorf("http error, status code %d", resp.StatusCode)
	}

	return nil
}

package vm

import (
	"fmt"

	log "github.com/sirupsen/logrus"
)

// VM represents the state of the virtual machine.
type VM struct {
	Stack   []Val       `json:"stack"`
	Env     map[Key]Val `json:"env"`
	Ip      uint        `json:"ip"`
	Program []Instr     `json:"program"`

	yielded bool `json:"-"`
}

func NewVM(program []Instr) *VM {
	return &VM{
		Env:     make(map[Key]Val),
		Program: program,
	}
}

func (vm *VM) UnmarshalJSON(data []byte) error {
	obj := make(map[string]interface{})
```

```
		stack, ok := (obj["stack"]).([]int)
		if !ok {
			return fmt.Errorf("invalid type stack")
		}
		for _, v := range stack {
			vm.Stack = append(vm.Stack, Val(v))
		}

		env, ok := obj["env"].(map[Key]Val)
		if !ok {
			return fmt.Errorf("invalid type env")
		}
		vm.Env = env

		ip, ok := obj["ip"].(uint)
		if !ok {
			return fmt.Errorf("invalid type ip")
		}
		vm.Ip = ip

		program, ok := obj["program"].([]map[string]interface{})
		if !ok {
			return fmt.Errorf("invalid type program")
		}

		for _, instr := range program {
			i, err := decodeObj(instr)
			if err != nil {
				return err
			}
			vm.Program = append(vm.Program, i)
		}

		return nil
}

func (vm *VM) Execute() error {
		log.WithFields(log.Fields{
			"ip":           vm.Ip,
			"stack_size":   len(vm.Stack),
			"program_size": len(vm.Program),
		}).Info("Executing a program")

		for ; vm.Ip < uint(len(vm.Program)); vm.Ip++ {
			err := vm.Program[vm.Ip].exec(vm)
			if err != nil {
				log.Errorf("%v\n", vm)
				return err
			}

			if vm.yielded {
				vm.Ip++
				return nil
			}
		}

		return nil
}

func (vm *VM) Yielded() bool {
		return vm.yielded
}

func (vm *VM) push(val Val) {
		vm.Stack = append(vm.Stack, val)
}
func (vm *VM) putEnv(name Key, val Val) error {
		vm.Env[name] = val
		return nil
}

func (vm *VM) getEnv(name Key) (Val, error) {
		if _, ok := vm.Env[name]; !ok {
			return 0, fmt.Errorf("value for name `%s` does not exist in the env", name)
		}
		return vm.Env[name], nil
}

func (vm *VM) pop() (Val, error) {
```

```
	if len(vm.Stack) == 0 {
		return Val(0), fmt.Errorf("there is nothing to pop from stack")
	}

	var val Val
	val, vm.Stack = vm.Stack[len(vm.Stack)-1], vm.Stack[:len(vm.Stack)-1]
	return val, nil
}

func (vm *VM) yield() {
	vm.yielded = true
}
```

ДОДАТОК В

Технологія взаємодії застосунків MATLAB і C# у комплексі моделювання гідроакустичних процесів

Опис програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ51181_18Б

Аркушів 9

Київ 2019

АНОТАЦІЯ

Даний додаток містить опис програмної системи розподіленої віртуальної машини. Створений програмний застосунок дозволяє виконувати довільний код род у віртуальному розподіленому середовищі. Код програми поділений на наступні пакети:

- assemble – ассемблерний модуль
- vm – віртуальна машина
- protocol – протокол розподіленої системи
- main – основний пакет

Отриманий програмний продукт спілкується по мережі використовуючи протокол HTTP. Дані до системи надсилаються за допомогою запиту на адресу /assemble.

Програмний продукт написаний мовою Go у середовищі GoLand.

ЗМІСТ

ЗАГАЛЬНІ ВІДОМОСТІ

У цьому додатку міститься опис реалізації розподіленої віртуальної машини. У додатку Б міститься програмний код головних модулів розроблюваних програмних систем.

Додатки можна використовувати у тестовому середовищі на будь-якій операційній системі, що містить інструментарій Docker. Для запуску реальної мережі необхідно збудувати проект для необхідних операційної системи та архітектури процесора.

При розробці програмного додатку використовувалась мова Go з використанням двох середовищ:

1. GoLand
2. Docker

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Розроблені компоненти компілюють довільний ассемблерний код, направляють його на виконання до віртуальної машини та розповсюджують стан між вузлами де необхідно.

Також розроблені інструменти для створення тестової мережі з кількох агентів у межах однієї ЕОМ.

Функціональні обмеження використання продукту співпадають лише з обмеженнями вбудованої мови ассемблер.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Програмний додаток ділиться на компоненти, які називають пакетами. Кожен пакет містить реалізацію певного функціонального елементу. Перелік пакетів:

- assemble – включає в себе лексичний та синтаксичний аналізатори вихідного тексту програми та компілятор
- vm – містить структуру віртуальної машини, її інтерфейс та реалізацію інструкцій віртуальної машини
- protocol – містить реалізацію сервера та клієнта, що використовуються для спілкування між вузлами
- main – містить реалізацію консольного інтерфейсу

Кожен пакет можна використовувати у сторонніх додатках,

ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Для забезпечення великої швидкості та ефективності виконання програми використану мову програмування Go, що дозволяє створювати паралельний код та синхронізувати його за допомогою каналів. Для відладки та форматування коду використано інструменти gofmt, goreturns та godebug.

В якості інтегрованого середовища розробки використано програмне забезпечення GoLand, що надає функціонал синтаксичного виділення коду, вбудованих інструментів для оптимізації та рефакторинку коду.

Для створення тестових мереж використано технологію контейнеризації інструментами Docker та Docker-Compose.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Для виклику програмного застосунку у тестовому серидовищі можна використати інструмент docker-compose, викликавши консольну команду docker-compose up.

Щоб завантажити продукт на реальній мережі необхідно запустити програму dvm.exe на всіх вузллах системи.

ВХІДНІ І ВИХІДНІ ДАНІ

Вхідними даними для розроблених додатків є ассемблерний код, що необхідно виконати.

Вихідними даними є результат виконаного коду, в залежності від предоставленої програми.